1 The asymptotic analysis of algorithms

Recall that in algorithm analysis and design, we are mainly interested in (i) algorithms that are *correct*, and (ii) algorithms that are *efficient*. The first of these is a binary statement. An algorithm is either correct, or it is not. Efficiency, however, is a relative term: algorithm $\mathcal{A}$ can be more or less efficient than algorithm $\mathcal{B}$.

In order to formally compare the resource usage (time or space) of different algorithms, we use *asymptotic analysis* and *asymptotic notation*. The "asymptotic" here means that we are only interested in the functional form of the resource usage as the size of the input n goes to infinity. This allows us to state formally and precisely how much of a resource (time or space) an algorithm uses, and to make statements like $\mathcal{A}$ is more, less, or equally efficient than $\mathcal{B}$.

For instance, if $T(n)$ is the running time of an algorithm when the input has size n, then we want to know the $g(n)$ such that

$$\lim_{n\to\infty} T(n) = \Gamma(g(n)) \ ,$$

where $g(n)$ is typically one of a small number of simple functional forms, and Γ indicates the kind of asymptotic relationship, typically O, Θ, or Ω (more about these below). Common examples of $g(n)$ include:

$g(n)$		meaning
1	:	constant resource usage, independent of n
$\log n$	:	sublinear, specifically logarithmic resource usage
n^c for $c < 1$	:	sublinear resource usage
n	:	linear resource usage
$n \log n$	:	super-linear, but much less than quadratic
n^c for $c > 1$	:	polynomial resource usage, super-linear
c^n for $c > 1$	:	exponential resource usage
n^n	:	extremely fast-growing resource usage

1.1 An example, and the tight bound of Θ

For instance, how much space does an array consume? In a low-level language like `C`, an array is literally a contiguous block of memory and thus takes an exact amount of space. If an element in the array takes up 32 bits of space, then an array of n elements takes $32n$ bits or $4n$ bytes. If an element takes 64 bits or 128 bits, then an n element array takes $64n$ or $128n$ bits of space.

Notice, however, that each of these has a common property: each takes space that grows exactly proportional to a *linear* function of the size of the array n. This property is true no matter how

many bits each element takes, so long as an element's "size" does not itself vary with n. This is precisely what we mean by saying that, in this case, the amount of space required by an array is $\Theta(n)$.

Formally, $\Theta(n)$ means something very specific. When we say that the amount of resources consumed (time or space) is some function $f(n)$ and that $f(n) = \Theta(g(n))$, we mean that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \exists c_1 > 0, \exists c_2 > 0, \forall n \geq n_0 > 0 \ . \tag{1}$$

In words, saying $f(n) = \Theta(g(n))$ is a mathematical claim that there exists some choice of input size n_0 such that for all larger inputs $n > n_0$, there are choices of the constants c_1, c_2 so that the function $g(n)$ is both an upper *and* a lower bound on $f(n)$. (See Figure 1a.)

A tight bound. When the upper- and lower-bound functions have the same functional form, we call the bound *tight*, which is what Θ means in algorithms. If you are asked to justify a claim that some function is $\Theta(g(n))$, one way to do this is to identify the constants c_1, c_2, n_0 that makes the statement true.

This way of comparing functions is useful in algorithms because it lets us focus on the dominant or leading-term behavior. It allows us to quickly compare different algorithms and make judgements about which will, in the long run, be more efficient, i.e., use fewer resources. For instance, $\Theta(n)$ is more efficient than $\Theta(n^2)$, which is more efficient than $\Theta(c^n)$ for $c > 1$, etc.

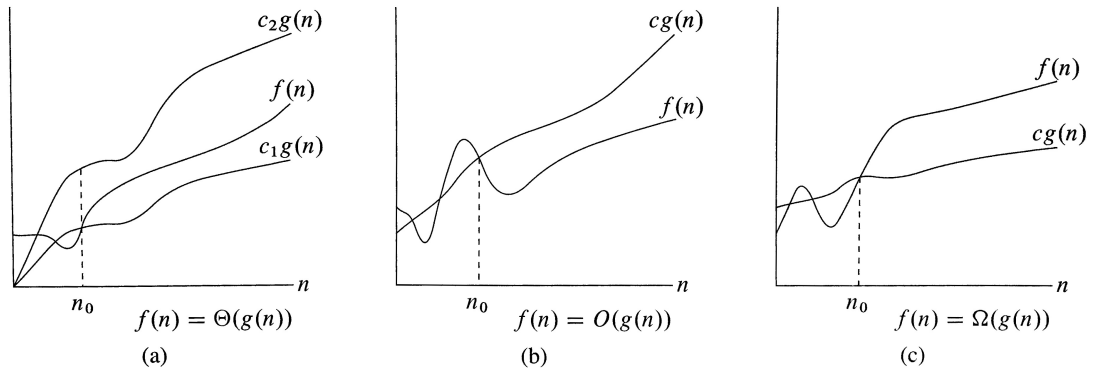

Figure 1: Schematics of the asymptotic growth of functions, showing (a) a tight bound $f = \Theta(g(n))$, (b) an upper bound $f = O(g(n))$, and (c) a lower bound $f = \Omega(g(n))$, for input sizes $n > n_0$. (Reproduced from *Introduction of Algorithms*, by Cormen et al.)

1.2 Upper or lower bounds: O and Ω

Tight bounds are the goal of all algorithm analysis: they tell us exactly how fast a function grows. But, tight bounds are often hard to obtain and instead we must settle for a weaker statement.

The most common of these is an **upper bound** or $O(g(n))$ or "Big-O", which relaxes the definition of Θ by omitting the claim that $g(n)$ is a lower bound on the resource usage.

Formally, $f(n) = O(g(n))$ means

$$f(n) \leq c_2 g(n) \qquad \exists c_2 > 0, \forall n \geq n_0 > 0 \ . \tag{2}$$

In words, O (or sometimes $\mathcal{O}$) says that our function $f(n)$ grows no faster than $g(n)$, and thus $g(n)$ provides an upper bound on the resource use. This is by far the most common type of statement we will make about an algorithm.

On the other hand, if we relax the definition of Θ to omit the upper bound, then we obtain a **lower bound** or $\Omega(g(n))$ or "Big-Ω", defined as

$$c_1 g(n) \leq f(n) \qquad \exists c_1 > 0, \forall n \geq n_0 > 0 \ . \tag{3}$$

Lower bounds on resource usage are substantially less useful than upper bounds, because lower bounds say nothing about the worst case scenario.

1.3 Strict bounds: o and ω

You may have noticed that in the definitions of Θ, O, and Ω, we use the $\leq$ symbol, meaning that $f(n)$ may grow exactly like $g(n)$.

This is an important detail: if we remove the equality from the definitions of O and Ω in Eqs. (2) and (3), we obtain *strict* bounds, which we denote by o and ω respectively. For instance, $n = \omega(1)$ and $n^2 = o(n)$. These bounds will be less common in this class, but often appear in more advanced algorithms topics.

1.4 A warning, and two examples of sloppy thinking

Asymptotic analysis is a powerful way to make precise statements about the way different functions grow with n, but with great power comes great responsibility.[1] A claim that some function is $\Theta(n)$ may be obvious, but you should be prepared to back up that claim by providing (or knowing that you can provide with a moment of work) the corresponding constants.

[1] Yes, you read that correctly.

It is easy to think of asymptotic notation, i.e., O, Θ, and Ω as heuristics. This can be dangerous. For instance, what is wrong with the following statement?

$$f(n) \text{ is at least } O(g(n))$$

The phrase "at least" is a lower-bound-type statement, while O is an upper-bound-type statement, so the statement overall is a contradiction: no function can grow at least as fast as a function that it grows no faster than.

Here is another problematic statement:

$$2n = O(2^n)$$

Technically, this statement is correct because all exponential functions grow faster than all linear functions. (Can you prove this?[2]) Here, we can derive the specific values of c_2 and n_0 that makes this statement true:

$$c_2 = 1 \qquad n_0 > 1 \ .$$

To verify this, we substitute these values into the original form: $2 \cdot 1 \leq 1 \cdot 2^1$, which is true. But, this statement is not very helpful because the exponential function grows much, much more quickly than the linear function. In fact

$$n = O(n^2) = O(n^3) = O(1.1^n) = O(1.11^n) = O(2^n) \ , \tag{4}$$

which illustrates that there is a lot of room (in fact, infinitely much) between $O(n)$ and $O(2^n)$. So, while technically correct, this bound is so loose as to be nearly trivial. In algorithms, it is a virtue to strive for the tightest bound possible. Ideally, we provide a tight bound Θ. Lacking that, we provide a tight upper O or tight lower bound Ω, depending on what we are trying to show about the algorithm.

1.5 Asymptotic analysis in practice

We will use asymptotic analysis and asymptotic notation throughout the semester. The final statements we are aiming for are compact Big-O statements. There are two basic strategies for formally arriving at a correct asymptotic statement.

1. Identify the constants n_0 and c_1 or c_2 for Ω or O (or both for Θ), and demonstrate that the formal definitions given in Eqs. (1), (2), and (3) are satisfied by these constants. Or,

2. Use asymptotic analysis, typically by applying L'Hopital's rule in the limit of $n \to \infty$, in order to obtain the correct functional relation.

[2]Hint: compare $f(n) = n^a$ and $g(n) = b^n$, for $a \geq 0$ and $b > 1$.

Recall L'Hopital's rule, which states that the asymptotic relationship of $f(n)$ to $g(n)$ is the same as the asymptotic relationship of the derivatives of $f(n)$ and $g(n)$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{\partial}{\partial n} f(n)}{\frac{\partial}{\partial n} g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \ .$$

When one applies L'Hopital's rule, the limit of the resulting ratio will itself be a function, and the form of this function tells us unambiguously whether $f(n)$ grows faster, slower, or as quickly as $g(n)$. Specifically,

$$\begin{aligned}
&\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \qquad \text{where } c \text{ is some } \textit{nonzero positive} \text{ constant} && \implies f(n) = \Theta(g(n)) \\
&\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty && \implies f(n) = \Omega(g(n)) \\
&\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 && \implies f(n) = O(g(n)) \ .
\end{aligned}$$

For instance, consider $f(n) = n^2 + n - 1$ and $g(n) = n^2$:

$$\begin{aligned}
\lim_{n\to\infty} \frac{n^2+n-1}{n^2} &= \lim_{n\to\infty} \frac{\frac{\partial}{\partial n}(n^2+n-1)}{\frac{\partial}{\partial n} n^2} \\
&= \lim_{n\to\infty} \frac{2n+1}{2n} \\
&= 1 \\
&\implies f(n) = \Theta(g(n)) \ .
\end{aligned}$$

Or, $f(n) = n\log n + n$ and $g(n) = n^2$:

$$\begin{aligned}
\lim_{n\to\infty} \frac{n\log n+n}{n^2} &= \lim_{n\to\infty} \frac{\frac{\partial}{\partial n}(n\log n+n)}{\frac{\partial}{\partial n} n^2} \\
&= \lim_{n\to\infty} \frac{2+\log n}{2n} \\
&= 0 \\
&\implies f(n) = O(g(n)) \ .
\end{aligned}$$

And, finally $f(n) = 2^n$ and $g(n) = 1.1^n$:

$$\begin{aligned}\lim_{n\to\infty} \frac{2^n}{1.1^n} &= \lim_{n\to\infty} \frac{\mathrm{e}^{(\ln 2)n}}{\mathrm{e}^{(\ln 1.1)n}} \\ &= \lim_{n\to\infty} \mathrm{e}^{n(\ln 2 - \ln 1.1)} \\ &= \lim_{n\to\infty} \mathrm{e}^{n \ln(2/1.1)} \\ &= \infty \\ &\implies f(n) = \Omega(g(n)) \ ,\end{aligned}$$

which did not require applying L'Hopital's rule, but did require some simple mathematical identities from the following useful list:

1. $(x^y)^z = x^{yz}$
2. $x^y x^z = x^{y+z}$
3. $\log_x y = z \implies x^z = y$
4. $x^{\log_x y} = y$ by definition
5. $\log(xy) = \log x + \log y$
6. $\log(x^c) = c \log x$
7. $\log_c(x) = \log x / \log c$

1.6 Atomic operations and analyzing code

Real algorithms use real resources, either time or space. Recall that in algorithm analysis, we count only *atomic operations* (time) and *atomic data structures* (space). What counts as atomic is given by our abstract model of computation, which in this class is the `RAM` model, where atomic operations include all the basic mathematical operators, including addition, subtraction, multiplication and division, along with comparison operations involving built-in variables (integers, reals, and characters), and storing a value in such a variable. Similarly, atomic data types are integers, reals, booleans, and characters.

When we analyze an algorithm, we count up how the number of atomic operations varies as a function of the input size n. However, we do not need to count *all* the operations, only those that vary in the same was as the asymptotic resource usage of the algorithm. This function is always dominated by the fastest-growing term. An example will illustrate this more clearly.

Consider the following non-recursive code snippet. Many algorithms are represented using recursive functions, in part because recursion is a powerful abstraction technique that allows complex operations to be expressed compactly. We will revisit asymptotic analysis for recursive functions later.

Loops like the following snippet, which is a simple accumulator function, can be very easy to analyze. This function takes as input a positive integer n and outputs the sum of the first n integers:[3]

```
computeSum(int n) {
    if n < 1 { return 0 }   // catch negative values!
    acc = 0                 // initialize the sum
    for i = 1; i<=n; i++ {  // add it up
        acc += i
    }
    return acc              // return the sum
}
```

How many atomic operations does this function compute? To illustrate the utility of asymptotic analysis, we will count them all and then make an observation.

First, the function is passing n by value, which is a single copy (count=1). It then makes a comparison (count=2) followed by an assignment (count=3). Now, we enter a loop. Setting up the loop takes an assignment followed immediately by a comparison, to check the loop termination criterion (count=5). Simple stuff so far.

Now, each pass around the loop incurs exactly the same costs: an increment of the loop counter, a comparison to check for termination, an addition and an assignment, which increases our count by 4. And, we pass exactly n times around this loop, before making a final copy operation. Thus, the total, exact count of atomic operations is $5 + 4n + 1$, where the first term counts the startup costs, the second counts the loop's costs and the third counts the exit costs.

Thus, asymptotically, this function takes $\Theta(n)$ time to compute the sum. The constants in the exact count would change if we added more operations to the startup (e.g., reading the complete works of Shakespeare),[4] or if we added more operations to each pass through the loop (e.g., solving every possible 9x9 Sudoku puzzle).[5] Sometimes, these constants can be very large, but asymptotically they are all equivalently unimportant.

[3] This function is not particularly efficient, as there is a simple mathematical formula for computing its result in $O(1)$ time. But, in your lives writing code, you will encounter many people who are less clever than you, and they will write slower versions of something you know how to do faster. And, sometimes, you will be the less clever person.

[4] The complete works of Shakespeare have a fixed size, and so reading them takes a fixed amount of time. The time required to debate their meaning, however, is unbounded.

[5] Also a task that with fixed size, albeit a very large one, because the size of the puzzle is fixed.

Suggestions for analyzing code. As you become more comfortable thinking about time or space and looking at code, many algorithms will immediately reveal their running times or space requirements to you.

For example, you might observe that an algorithm is a pair of nested `for` loops each of length n, and the interior of these loops cost some constant amount of work. Thus, you would conclude that the algorithm takes $\Theta(n^2)$ time. Further, you might observe that these loops only ever allocate a constant number of variables, and thus the algorithm takes $\Theta(1)$ space.

Other algorithms, however, will be non-trivial to understand: their behavior at a particular step will depend on the sequence of decisions made before. The key idea for analyzing these algorithms, which include most of the algorithms in this class, is to identify the abstract pattern of the algorithm's performance. Identifying a property of the algorithm or data structure that is constant as it progresses is the common element in all algorithm problems. Or, lacking that, try to bound the algorithm's behavior by making it do more work, e.g., making a loop run from 1 to n rather than from some i s.t. $1 < i < n$ to n.

2 On your own

1. Read Sections 5.1 and 5.2 in Kleinberg–Tardos.

3 Starting Divide & Conquer Algorithms

One strategy for designing efficient algorithms is the "divide and conquer" approach, which is also called, more simply, a recursive approach. The analysis of recursive algorithms often produces mathematical equations called *recurrence relations*. It is the analysis of these equations that produces the bound on the algorithm running time.

The divide and conquer strategy has three basic parts. For a given problem of size n,

1. **divide**: break a problem instance into several smaller instances of the same problem
2. **conquer**: if a smaller instance is trivial, solve it directly; otherwise, divide again
3. **combine**: combine the results of smaller instances together into a solution to a larger instance

That is, we split a given problem into several smaller instances (usually 2, but sometimes more depending on the problem structure), which are easier to solve, and then combine those smaller solutions together into a solution for the original problem. The goal is to keep dividing until we get the problem down to a small enough size that we can solve it quickly (or trivially). Fundamentally,

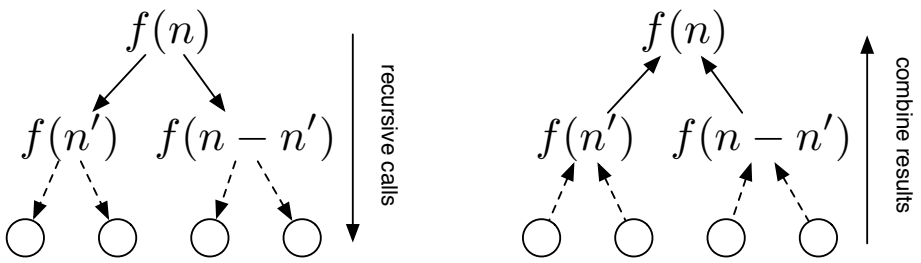

Figure 2: Schematic of the divide & conquer strategy, in which we recursively divide a problem of size n into subproblems of size n' and $n - n'$, until a trivial case is encountered (left side). The results of each pair of subproblems are combined into the solution for the larger problem. The computation of any divide & conquer algorithms can thus be viewed as a tree.

divide and conquer is a recursive approach, and most divide and conquer algorithms have the following structure:

```
function fun(n) {
   if n==trivial {
      solve and return
   } else {
      partA = fun(n')
      partB = fun(n-n')
      AB    = combine(A,B)
      return AB
   }
}
```

The recursive structure of divide and conquer algorithms makes it useful to model their asymptotic running time $T(n)$ using recurrence relations, which often have the general form

$$T(n) = a\,T(g[n]) + f(n) \ , \tag{5}$$

where a is a constant denoting the number of subproblems we break a given instance into, $g[n]$ is a function of n that describes the size of the subproblems, and $f(n)$ is the time required to combine the smaller results / divide the problem into the smaller versions. There are several strategies for solving recurrence relations we'll cover in the Divide & Conquer unit (staring today, continuing next week): the "unrolling" method and the master method; other methods include annihilators, changing variables, characteristic polynomials and recurrence trees. You can read about many of these in the textbook (Section 5.2).

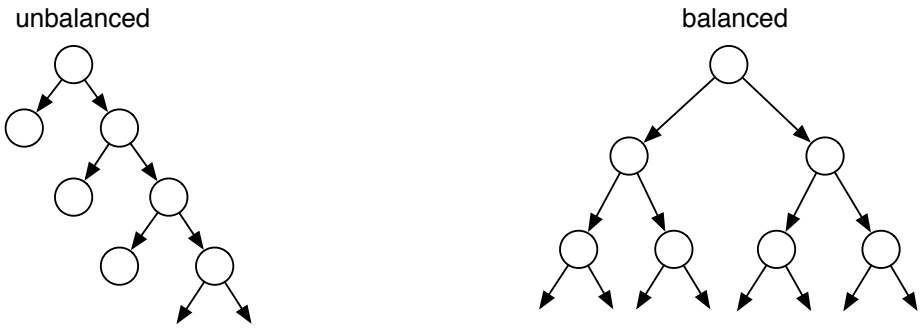

Figure 3: Examples of unbalanced and balanced binary trees. A fully unbalanced tree has depth $\Theta(n)$ while a full balanced tree has depth $\Theta(\log n)$. As we go through the QuickSort analysis next week, keep these pictures in mind, as they will help you understand why QuickSort is a fast algorithm.

Consider the case of $a = 2$. Figure 1 shows a schematic of the way a generic divide and conquer algorithm works. The algorithm effectively builds a computation or recursion tree, in which an internal node represents a specific non-trivial subproblem. Trivial or base cases are located at the leaves. As the function explores the tree, it uses a *stack* data structure (CLRS Chapter 10.1) to store the previous, not-yet-completed problems, to which the computer will return once it has completed the subproblems rooted at a given internal node. The path of the calculation through the recursion tree is a depth-first exploration.

The division of a problem of size n into subproblems of sizes $n - n'$ and n' determines the depth the tree, which determines how many times we incur the $f(n)$ cost. The sum of these costs is the running time. Consider the cases of $n' = 1$ and $n' = n/2$ (see Figure 2). In the first case, each time we recurse, we carve off a trivial case from the current size, and this produces highly unbalanced recursion trees, with depth n. In the second case, we divide the problem in half each time, and the depth of the tree is $\lg n$. If $f(n) = \omega(1)$, then the total cost is different between the two cases; in particular, the more balanced case is lower cost.